Async with Fibers

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

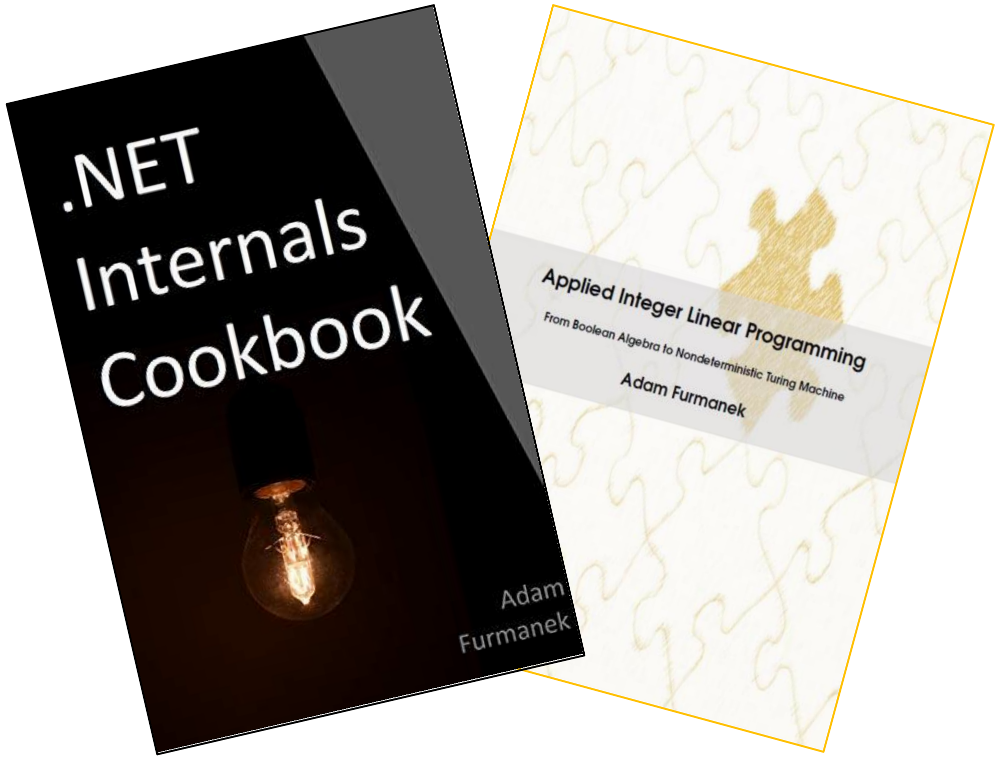
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam


.NET Internals Cookbook
Adam Furmanek
Applied Integer Linear Programming
From Boolean Algebra to Nondeterministic Turing Machine
Adam Furmanek

Random IT Utensils

IT, operating systems, maths, and more.

Agenda

Why async.

OS Thread vs Others.

Demos.

Why async

Asynchronous Programming Model (APM)

BeginOperation returns an object implementing *IAsyncResult*.

- Triggers the asynchronous calculations on different thread.
- Can also accept a callback to be called when the operation is finished.

IAsyncResult:

- Has some *AsyncState*.
- Contains *WaitHandle* which we can use to block the application.
- Has flag indicating whether the operation is completed.

EndOperation accepts *IAsyncResult* as a parameter and returns the same as synchronous counterpart.

- Throws all exceptions if needed.
- If the operation hasn’t finished, blocks the thread.

```
var fs = new FileStream(@"C:\file.txt");
byte[] data = new byte[100];
fs.BeginRead(data, 0, data.Length,
(IAsyncResult ar) =>
    {
        int bytesRead = fs.EndRead(ar);
        fs.Close();
    }, null
);
```

Event-based Asynchronous Pattern (EAP)

MethodNameAsync.

- Triggers the operation on a separate thread.

MethodNameCompleted.

- Event fired when the operation finishes.
- Passes parameter *AsyncCompletedEventArgs*.

AsyncCompletedEventArgs:

- Contains flag if the job was cancelled.
- Contains all the errors.
- Has some *UserState*.

Can be canceled.

Can be used easily with *BackgroundWorker*.

```
backgroundWorker.DoWork += backgroundWorker_DoWork;

private void backgroundWorker_DoWork(object sender,
DoWorkEventArgs e)

{

    // ...

}

private void
backgroundWorker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)

{

    // ...

}
```

Task-based Asynchronous Pattern (TAP)

Task.Run accepting delegate triggers the job:

- Equivalent to *Task.Factory.StartNew(job, CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);*
- Unwraps the result if needed (so we get *Task<int>* instead of *Task<Task<int>>*).

Task can be created manually via constructor and schedulled using *Start* method.

Can be joined by using *ContinueWith*.

Exceptions are caught and propagated on continuation.

Can be used with *TaskCompletionSource*.

Can be cancelled with *CancellationToken*.

Can report progress using *IProgress<T>.*

Parallel Language Integrated Queries (PLINQ)

Created when *AsParallel* called on *IEnumerable*. Can be reverted by *AsSequential*.

Operations defined in *ParallelEnumerable* class.

Can be ordered by calling *AsOrdered*.

Task merging can be configured by specifying *ParallelMergeOptions*.

Maximum number of concurrent tasks can be controlled using *WithDegreeOfParallelism*.

Parallelism is not mandatory! Can be forced with *ParallelExcecutionMode*.

Each *AsParallel* call reshuffles the tasks.

async and *await*

await can be executed on anything awaitable – not necessarily a *Task*!

- Task.Yield returns *YieldAwaitable*

Duck typing - awaitable type must be able to return *GetAwaiter()* with the following:

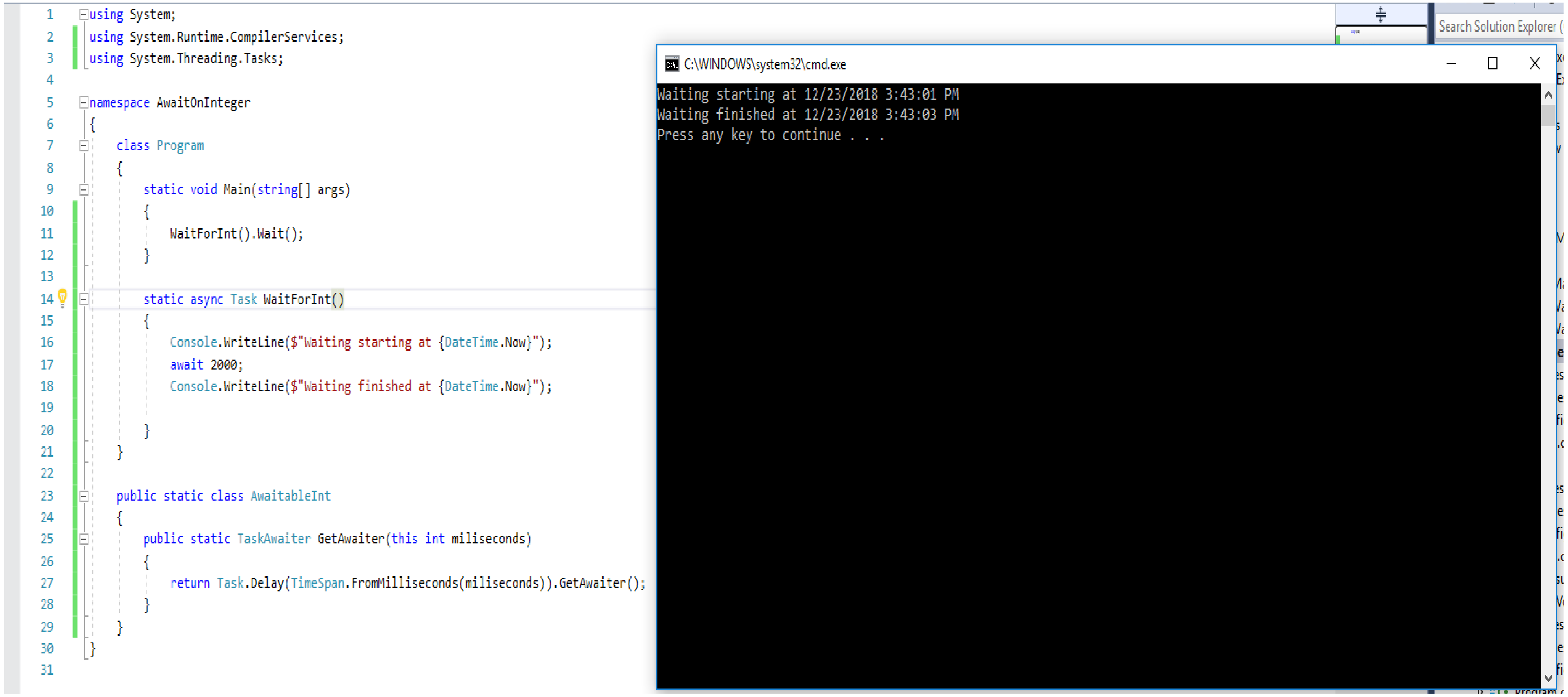
- Implements *INotifyCompletion* interface
- `bool IsCompleted { get; }`
- `void OnCompleted(Action continuation);`
- `TResult GetResult(); // or void`

async means nothing — it only instructs the compiler to create a state machine.

We can make any type awaitable using extension methods!

Very similar to *foreach*.

Awaiting on integer



```csharp
using System;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;

namespace AwaitOnInteger
{
    class Program
    {
        static void Main(string[] args)
        {
            WaitForInt().Wait();
        }

        static async Task WaitForInt()
        {
            Console.WriteLine($"Waiting starting at {DateTime.Now}");
            await 2000;
            Console.WriteLine($"Waiting finished at {DateTime.Now}");

        }
    }

    public static class AwaitableInt
    {
        public static TaskAwaiter GetAwaiter(this int miliseconds)
        {
            return Task.Delay(TimeSpan.FromMilliseconds(miliseconds)).GetAwaiter();
        }
    }
}
```

```
Waiting starting at 12/23/2018 3:43:01 PM
Waiting finished at 12/23/2018 3:43:03 PM
Press any key to continue . . .
```

Asynchronous code **does not block** the operating system level thread.

async in C#

async in C# is implemented as:

- coroutine compiler level transformation with
- service locator for promise orchestration and
- statically bound promise factories

async milestones

C# with Async CTP in 2011. Then with version 5 in 2012.

Haskell async package in 2012.

Python with version 3.5 in 2015.

TypeScript with version 1.7 in 2015.

JavaScript with ECMAScript 2017.

Rust with version 1.39.0 in 2019.

C++ with version 20 in 2020.

Java with version 17 with completely different approach – Loom.

OS Thread vs Others

Native thread

Two types: foreground and background (don’t stop application from terminating).

Consists of *Thread Kernel Object*, two stacks (user mode and kernel mode) and *Thread Environment Block* (TEB).

User mode stack by default has 1 MB, kernel mode has 12/24 KB.

Has impersonation info, security context, *Thread Local Storage* (TLS).

Windows schedules threads, not processes!

How many threads does the *notepad.exe* have?

Managed thread

Has ID independent of native thread ID.

Can have name.

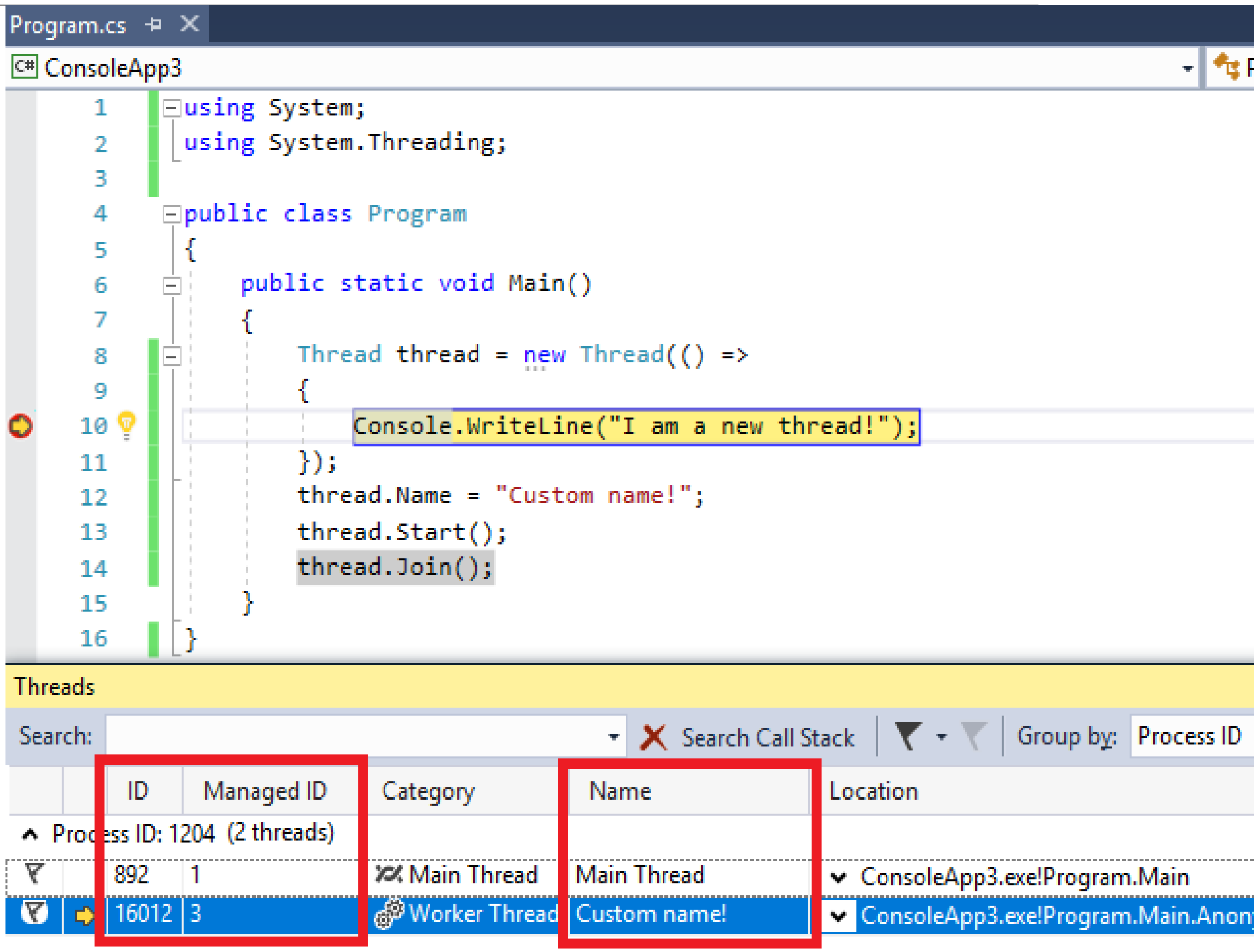
Can be suspended but this should not be done! Can be aborted by *Thread.Abort* but this doesn't guarantee anything.

Precommits stack when created.

Unhandled exception kills the application in most cases.

In .NET 1 it was different:

- Exception in other thread was printed to the console and the thread was terminated.
- Exception on the finalizer was printed to the console and finalizer was still working.
- Exception on the main thread resulted in application termination.



Program.cs

ConsoleApp3

```
using System;
using System.Threading;

public class Program
{
    public static void Main()
    {
        Thread thread = new Thread(() =>
        {
            Console.WriteLine("I am a new thread!");
        });
        thread.Name = "Custom name!";
        thread.Start();
        thread.Join();
    }
}
```

Threads

Search: Search Call Stack Group by: Process ID

ID	Managed ID	Category	Name	Location
Process ID: 1204 (2 threads)				
892	1	Main Thread	Main Thread	ConsoleApp3.exe!Program.Main
16012	3	Worker Thread	Custom name!	ConsoleApp3.exe!Program.Main.Anon

Green Threads or Virtual Threads

Scheduled by a runtime library or virtual machine. Not by the OS.

They run in user space.

The name comes from The Green Team at Sun Microsystems designing the original thread library for Java.

Green Threads

ADVANTAGES

Can be started much faster on some VMs.

They outperform native threads (YMMV) on synchronization.

DISADVANTAGES

On a multi-core procesor cannot assing work to multiple processors.

They underperform on I/O and context switching.

They must use asynchronous I/O operation – otherwise they block all green threads.

Other variations

Erlang uses Green Processors as they do not share data memory directly (that’s a simplification).

Go uses Goroutines which are run in virtual threads.

Julia uses green threads for tasks.

Stackless Python uses Tasklets.

Cpython uses Greenlets, Eventlets, Gevents.

Coroutines

Methods which can be suspended and resumed.

Used for event loops, iterators, Infinite lists, pipes.

Generalize subroutines.

They are cooperatively multitasked.

They are a language-level construct.

Generators are called Semicoroutines – Coroutine can control where execution continues after yield. One can be implemented with another by using Trampoline.

yield

```
using System;
using System.Collections;

class Test
{
    static IEnumerator GetCounter()
    {
        for (int count = 0; count < 10; count++)
        {
            yield return count;
        }
    }
}
```

Drawbacks

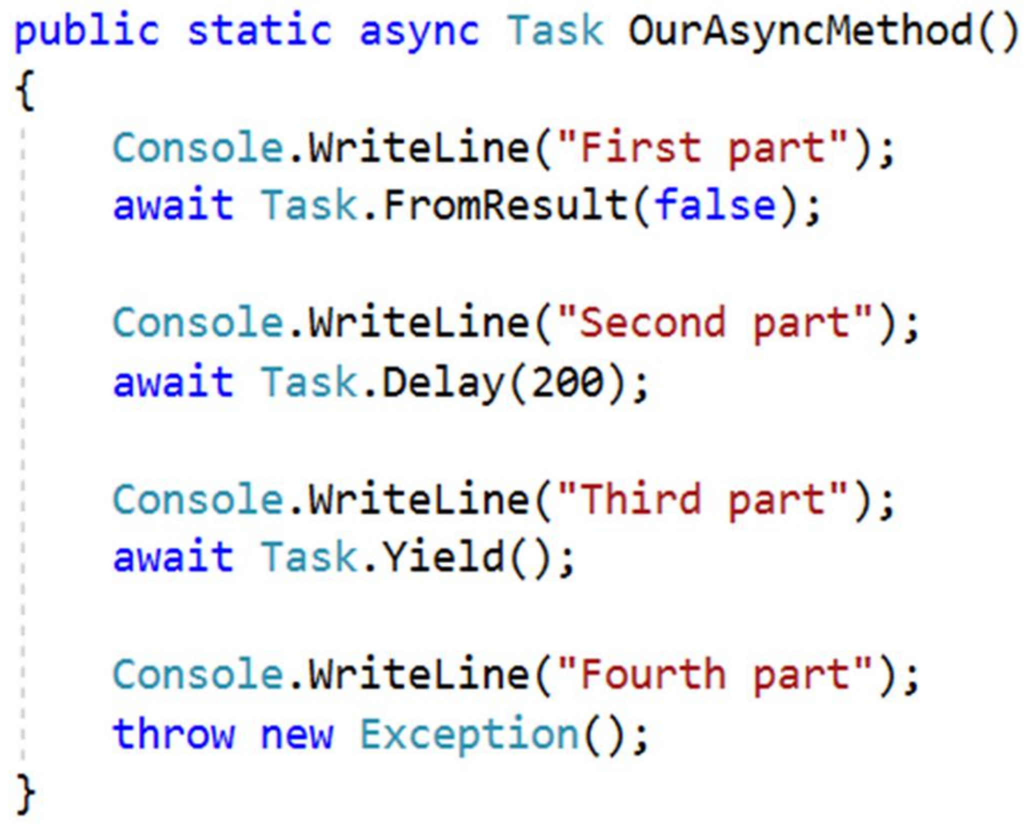
The introduce function colouring.

They can’t be used in constructors or *void* methods.

They may change exceptions handling mechanisms (as we exit function immediately and introduce *try+catch* blocks).

They increase memory allocation.

Async Method



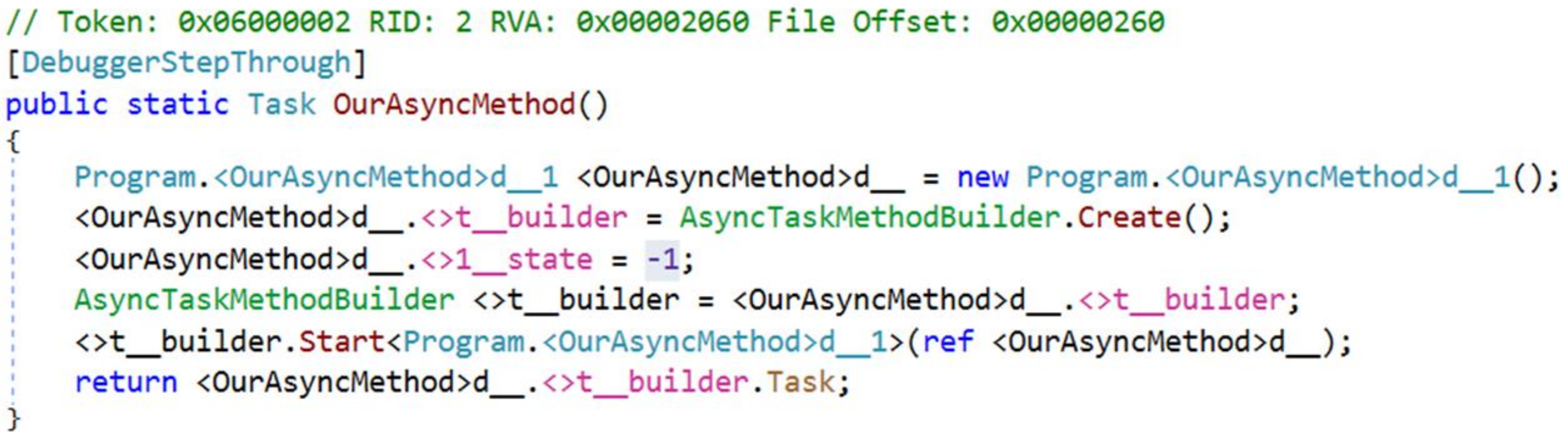
```
public static async Task OurAsyncMethod()
{
    Console.WriteLine("First part");
    await Task.FromResult(false);

    Console.WriteLine("Second part");
    await Task.Delay(200);

    Console.WriteLine("Third part");
    await Task.Yield();

    Console.WriteLine("Fourth part");
    throw new Exception();
}
```

Async Method after compilation



```
// Token: 0x06000002 RID: 2 RVA: 0x00002060 File Offset: 0x00000260
[DebuggerStepThrough]
public static Task OurAsyncMethod()
{
    Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = new Program.<OurAsyncMethod>d__1();
    <OurAsyncMethod>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
    <OurAsyncMethod>d__.<>1__state = -1;
    AsyncTaskMethodBuilder <>t__builder = <OurAsyncMethod>d__.<>t__builder;
    <>t__builder.Start<Program.<OurAsyncMethod>d__1>(ref <OurAsyncMethod>d__);
    return <OurAsyncMethod>d__.<>t__builder.Task;
}
```

Continuation Passing Style

Control is passed explicitly in the form of a continuation.

Function takes an extra argument- a function – and invokes it upon completion.

CPS

```
using System;

public class Program
{
    static int Fibonacci(int n)
    {
        switch (n)
        {
            case 0: return 0;
            case 1: return 1;
            default: return Fibonacci(n - 1) + Fibonacci(n - 2);
        }
    }

    public static void Main(string[] args)
    {
        Console.WriteLine($"{10}: {Fibonacci(10)}");
    }
}
```

```
using System;

public class Program
{
    static void FibonacciCps(int n, Action<int> continuation)
    {
        switch (n)
        {
            case 0: continuation(0); break;
            case 1: continuation(1); break;
            default:
                FibonacciCps(n - 1, fib1 =>
                    FibonacciCps(n - 2, fib2 =>
                        continuation(fib1 + fib2)));
                break;
        }
    }

    public static void Main(string[] args)
    {
        FibonacciCps(10, fib => Console.WriteLine($"{10}: {fib}"));
    }
}
```

call-with-current-continuation call/cc

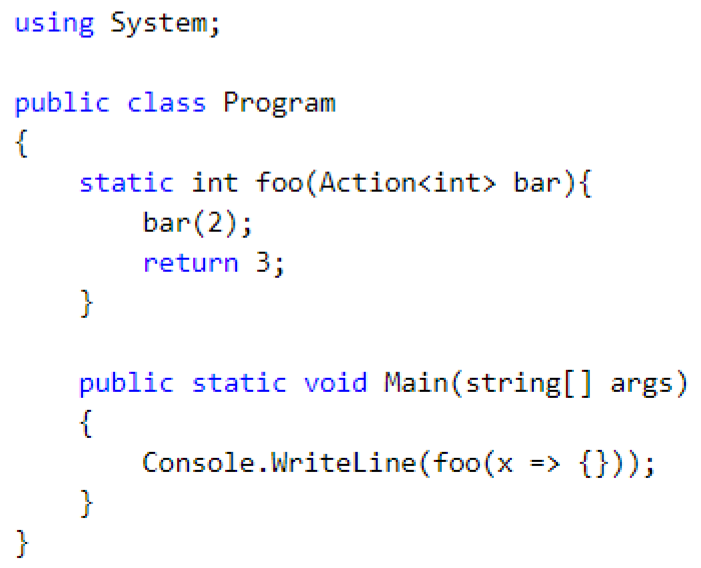
Takes one function as an argument: *call/cc f*

Calls *f* with current continuation of the expression.

Can be used to implement language constructs like return or loops.

Is considered unreadable.

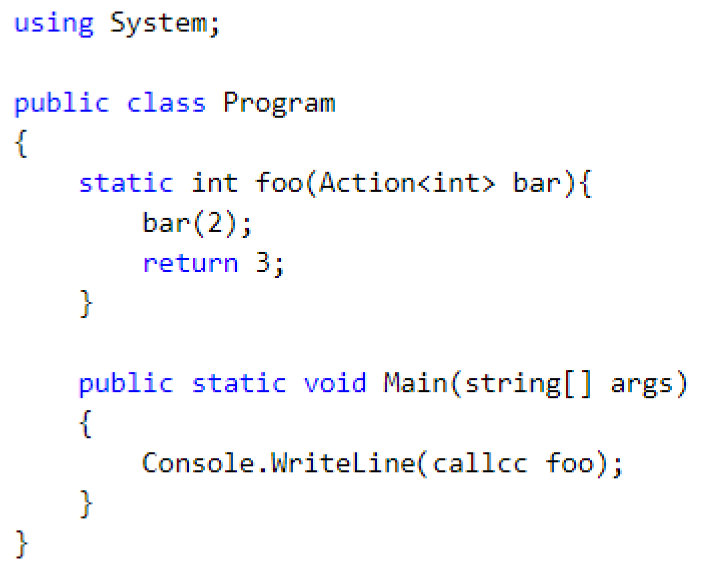
call/cc



```
using System;

public class Program
{
    static int foo(Action<int> bar){
        bar(2);
        return 3;
    }

    public static void Main(string[] args)
    {
        Console.WriteLine(foo(x => {}));
    }
}
```



```
using System;

public class Program
{
    static int foo(Action<int> bar){
        bar(2);
        return 3;
    }

    public static void Main(string[] args)
    {
        Console.WriteLine(callcc foo);
    }
}
```

Prints 3

Prints 2

Fibers

Lightweight threads with cooperative multitasking.

They must manually and explicitly yield control.

They are system-level construct. They may be viewed as an implementation of coroutines.

Supported by WinAPI.

Drawbacks

They may have a significant memory footprint (just like threads)

- There are tricks to minimize the memory by using guard page and dynamically allocating stack or by abandoning contiguous stacks and use segmented ones.
- Segmented stacks were abandoned due to the host-split problem where we call a method on a stack boundary and repeatedly allocate and deallocate the page.

Thread Local Storage cannot be reliably used.

Blocking call blocks all fibers.

They can’t be used with thread-based synchronization primitives.

Fibers

First, we need to convert a thread into a fiber with *ConvertThreadToFiber*

- This initializes OS structures.
- Method isn't directly exposed in C#.

Then, we create fibers with *CreateFiber*

- Allocates a fiber object.
- Assigns a stack.
- Sets up an execution to begin at the specified start address.
- This does not schedule the fiber.

Then, we can schedule fibers with *SwitchToFiber*

- Saves the state of the current fiber and restores the state of the specified fiber.

Demos

Coroutines

COROUTINE COMPILER LEVEL TRANSFORMATION

Fibers

DO NOT BLOCK THE OPERATING SYSTEM LEVEL THREAD

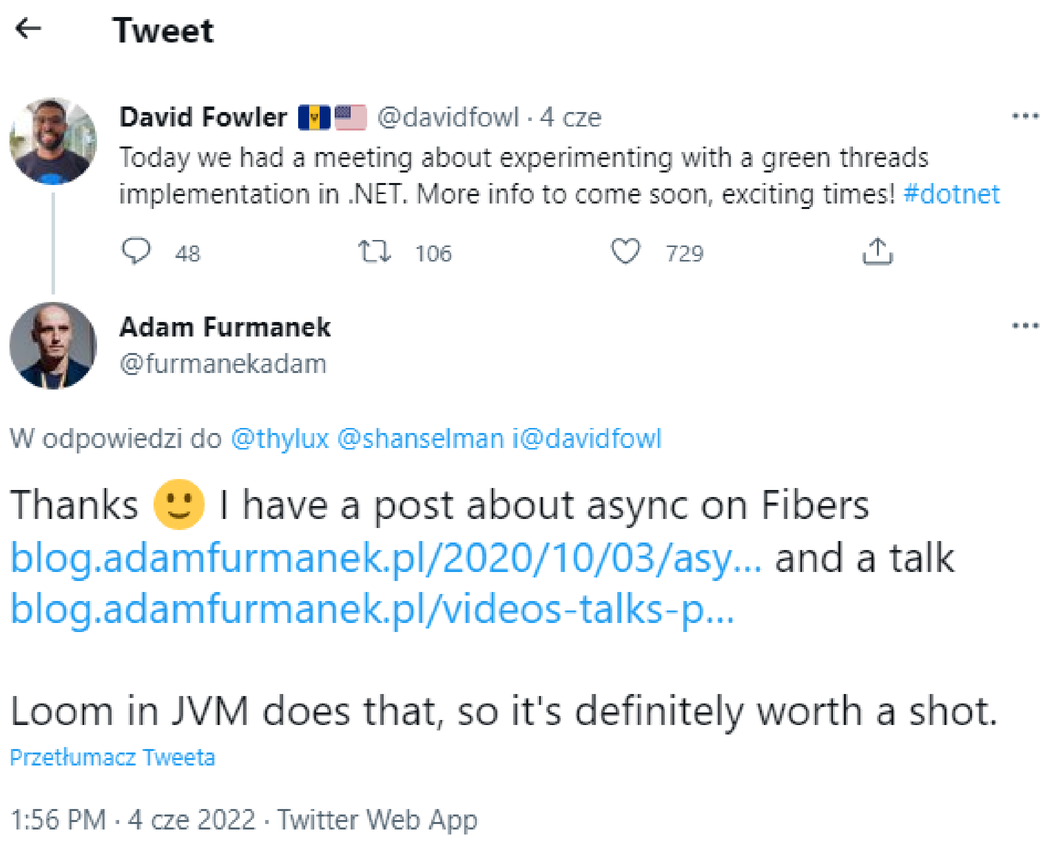
Monads

STATICALLY BOUND PROMISE FACTORIES

Generics

SERVICE LOCATOR FOR PROMISE ORCHESTRATION

But why?


Tweet

David Fowler @davidfowl · 4 cze
Today we had a meeting about experimenting with a green threads implementation in .NET. More info to come soon, exciting times! #dotnet

48 106 729

Adam Furmanek
@furmanekadam

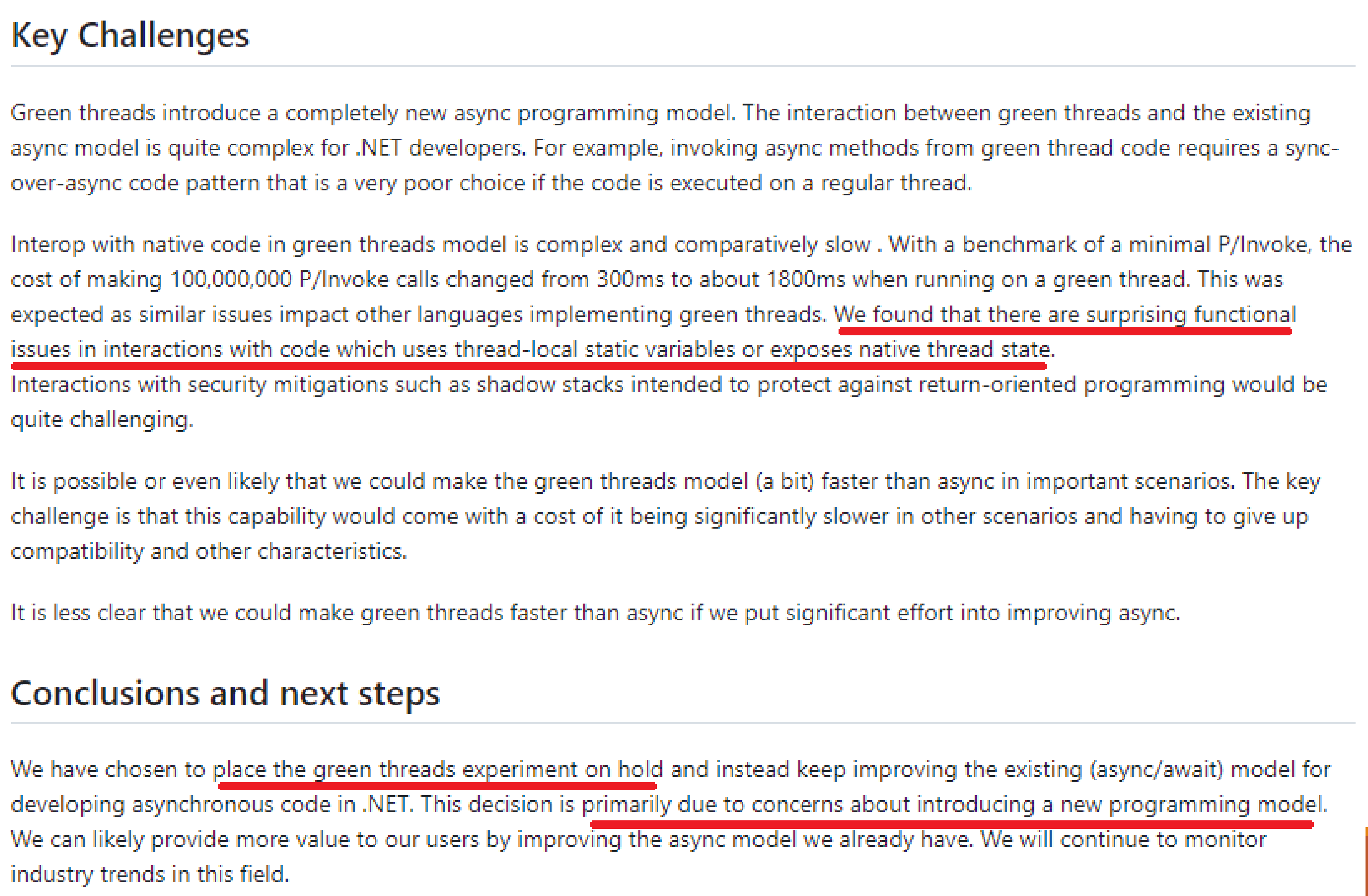
W odpowiedzi do @thylux @shanselman i@davidfowl

Thanks 🙂 I have a post about async on Fibers blog.adamfurmanek.pl/2020/10/03/asy... and a talk blog.adamfurmanek.pl/videos-talks-p...

Loom in JVM does that, so it's definitely worth a shot.

Przetłumacz Tweeta

1:56 PM · 4 cze 2022 · Twitter Web App



Key Challenges

Green threads introduce a completely new async programming model. The interaction between green threads and the existing async model is quite complex for .NET developers. For example, invoking async methods from green thread code requires a sync-over-async code pattern that is a very poor choice if the code is executed on a regular thread.

Interop with native code in green threads model is complex and comparatively slow . With a benchmark of a minimal P/Invoke, the cost of making 100,000,000 P/Invoke calls changed from 300ms to about 1800ms when running on a green thread. This was expected as similar issues impact other languages implementing green threads. We found that there are surprising functional issues in interactions with code which uses thread-local static variables or exposes native thread state. Interactions with security mitigations such as shadow stacks intended to protect against return-oriented programming would be quite challenging.

It is possible or even likely that we could make the green threads model (a bit) faster than async in important scenarios. The key challenge is that this capability would come with a cost of it being significantly slower in other scenarios and having to give up compatibility and other characteristics.

It is less clear that we could make green threads faster than async if we put significant effort into improving async.

Conclusions and next steps

We have chosen to place the green threads experiment on hold and instead keep improving the existing (async/await) model for developing asynchronous code in .NET. This decision is primarily due to concerns about introducing a new programming model. We can likely provide more value to our users by improving the async model we already have. We will continue to monitor industry trends in this field.

Summary

Coroutines do not integrate with platform that well.

Global state is inflexible.

Green threads can simplify things significantly and make code cleaner.

Green threads can’t be easily used with OS thread-based primitives.

Know your synchronization context.

Q&A

References

Jeffrey Richter - „CLR via C#”

Jeffrey Richter, Christophe Nasarre - „Windows via C/C++”

Mark Russinovich, David A. Solomon, Alex Ionescu - „Windows Internals”

Penny Orwick – „Developing drivers with the Microsoft Windows Driver Foundation”

Mario Hewardt, Daniel Pravat - „Advanced Windows Debugging”

Mario Hewardt - „Advanced .NET Debugging”

Steven Pratschner - „Customizing the Microsoft .NET Framework Common Language Runtime”

Serge Lidin - „Expert .NET 2.0 IL Assembler”

Joel Pobar, Ted Neward — „Shared Source CLI 2.0 Internals”

Adam Furmanek – „.NET Internals Cookbook”

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md — „Book of the Runtime”

https://blogs.msdn.microsoft.com/oldnewthing/ — Raymond Chen „The Old New Thing”

References

https://blog.adamfurmanek.pl/blog/2016/10/08/async-wandering-part-1/ — async in unit tests

https://blog.adamfurmanek.pl/blog/2017/01/07/async-wandering-part-3/ — WinForms

https://blog.adamfurmanek.pl/blog/2017/06/03/capturing-thread-creation-to-catch-exceptions/ — overriding Thread constructor to handle exceptions

https://blog.adamfurmanek.pl/blog/2017/01/14/async-wandering-part-4-awaiting-for-void-methods/ — awaiting *async void*

https://blog.adamfurmanek.pl/blog/2018/10/06/async-wandering-part-5-catching-exceptions-from-async-void/ — catching exceptions in *async void*

References

https://www.codeproject.com/Articles/662735/Internals-of-Windows-Thread - Windows threads

http://aviadezra.blogspot.com/2009/06/net-clr-thread-pool-work.html - .NET ThreadPool

https://mattwarren.org/2017/04/13/The-CLR-Thread-Pool-Thread-Injection-Algorithm/ — ThreadPool injection algorithm

http://www.microsoft.com/download/en/details.aspx?id=19957 — TAP

https://msdn.microsoft.com/en-us/magazine/gg598924.aspx?f=255&MSPPError=-2147217396 – It's all about the synchronization context

https://blogs.msdn.microsoft.com/seteplia/2018/10/01/the-danger-of-taskcompletionsourcet-class/ - TaskCompletionSource

https://blog.stephencleary.com/2014/04/a-tour-of-task-part-0-overview.html - Task internals

https://blog.stephencleary.com/2017/03/aspnetcore-synchronization-context.html — ASP.NET Core sychronization context

https://blogs.msdn.microsoft.com/pfxteam/2012/06/15/executioncontext-vs-synchronizationcontext/ — ExecutionContext internals

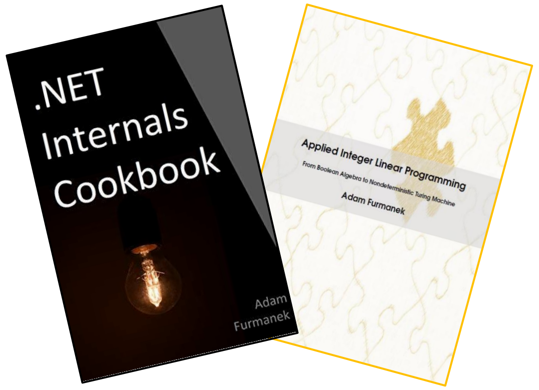
https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-3-runtime-context — ExecutionContext internals

https://blogs.msdn.microsoft.com/seteplia/2017/11/30/dissecting-the-async-methods-in-c/ - State machine

https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-1-compilation - State machine

https://github.com/dotnet/runtimelab/issues/2398 - .NET Green Thread experimentations

https://github.com/dotnet/runtimelab/blob/feature/green-threads/docs/design/features/greenthreads.md - .NET Green Thread Reports


.NET Internals Cookbook
Adam Furmanek
Applied Integer Linear Programming
From Boolean Algebra to Nondeterministic Turing Machine
Adam Furmanek

Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM